
# Recueil de sujets de TD M2101
# Programmation Bas Niveau

Université de BORDEAUX
Département Informatique
I.U.T. de Bordeaux

# TD 1 M 2 101
# Microprocesseur fictif

L'objectif de ce test est d'étudier une architecture VRISC. Il s'agit d'un microprocesseur virtuel appelé SIC (Single Instruction Computer). Le jeu d'instruction est donc très réduit car il n'en contient qu'une seule! Il s'agit de l'instruction Subtract and Branch if Negative:

**Sbn a,b,c** # M[a]=M[a]-M[b];
if (M[a]<0) go to c;

Remarque: si M[a]=0 on passe à l'instruction suivante (on passe à l'adresse qui suit celle où se trouve l'instruction Sbn a,b,c).

M[a] désigne le contenu de l'adresse a. Par simplification, on pourra prendre (sans que cela soit nécessaire ni obligatoire) M[a]=a.
SIC n'a pas de registre d'instruction ni d'autre instruction que Sbn.

exemple:
début: Sbn temp,temp,.+1
A l'adresse dont l'étiquette est "début:" figure l'instruction qui annule le contenu de l'adresse "temp".
".+1" désigne l'adresse de l'instruction qui suit celle présente à l'adresse "début:".
On suppose que "temp" est l'adresse d'un mot de mémoire utilisable pour des résultats temporaires.

**Questions**:

1) Donner la signification de l'abréviation VRISC.

2) Ecrire un programme SIC permettant de copier un nombre de l'adresse "a" vers l'adresse "b". On utilisera l'adresse "temp".

3) Donner un schéma fonctionnel de SIC sur la base de celui du microprocesseur fictif vu en cours.
Donner les différentes étapes d'exécution d'une instruction **Sbn a,b,c** en mettant en évidence le chemin qu'empruntent les donnés, adresses ou instructions qui sont traitées.
Quel est l'intérêt d'une telle structure par rapport au microprocesseur fictif? On se mettra dans l'hypothèse où l'adresse de l'instruction qui suit celle présente à l'adresse "début:", par exemple, est l'adresse « début + 3 ». Expliquez ce choix.

4) Ecrire un programme SIC ajoutant deux nombres a et b.

5) Que calcule le programme suivant:
c=0; while(b>0) {b=b-1; c=c+a;}

6) Ecrire un programme SIC multipliant a (le contenu de l'adresse a) par b (le contenu de l'adresse b), plaçant le résultat dans c.
**On suppose que l'adresse de mémoire un (1) contient le nombre 1**. On suppose que a et b sont des entiers positifs et qu'on peut les modifier.

7) Ecrire un programme SIC divisant a (le contenu de l'adresse a) par b (le contenu de l'adresse b), plaçant le résultat dans c et d. Pourquoi faut-il deux adresses pour le résultat? On suppose que l'adresse de mémoire un (1) contient le nombre 1. On suppose que a et b sont des entiers positifs et qu'on peut les modifier.

# Microprocesseur fictif : Machine A Pile TD3 M 2 101

L'objet de ce problème est l'étude d'une machine à Pile appelée MaP par la suite.

La machine considérée est essentiellement constituée :

- D'une mémoire de mots adressable par mots (32 bits) contenant des entiers (codés en complément à 2) ou des adresses,
- D'une pile de mots dont les adresses sont comprises entre B(ase) et L(imite) : le pointeur de pile, SP (Stack Pointer), qui contient l'adresse du sommet de la pile a donc une valeur telle que B<=SP<=L. Si SP<B (respectivement SP > L) alors un signal d' « underflow » (respectivement d' « overflow ») est émis,
- D'un « Registre de Condition », RC positionné par certaines opérations (par exemple les comparaisons).

**Question 1** : traduisez les termes signal d' « underflow » et signal d' « overflow »

**Question 2** : en s'inspirant des microprocesseurs traditionnels, fournir un schéma fonctionnel simple de la Machine à Pile (MaP). S'agit-il d'une machine de Von Neuman ? Dispose-t-elle d'un compteur ordinal (PC) ?

*Voici maintenant quelques instructions de MaP :*

| | |
|---|---|
| *push M* | *: empile (si possible) le contenu du mot d'adresse symbolique M, SP=SP+1* |
| *push =C* | *: empile (si possible) la constante (entière) C, SP=SP+1* |
| *pop M* | *: dépile un mot et le copie (si possible) dans le mot d'adresse M, SP=SP-1* |

**Question 3** : Précisez le fonctionnement (microprogramme) des instructions push et pop, justifiez les termes « si possible » utilisés dans le paragraphe précédent de description sommaire de ces instructions.

*Les opérations arithmétiques fondamentales agissent sur les deux éléments qui sont au sommet de la pile. Voici le microprogramme de « add » et « sub » :*
*add : dépile x, dépile y, r=x+y, empile r*
*sub : dépile x, dépile y, r=y-x, empile r*
*Où x,y et r sont des registres internes à l'UC non accessibles au programmeur.*

**Question 4** : donner le microprogramme des instructions « mul » et « div » en précisant la valeur de SP après exécution de chaque instruction, si celle-ci est possible.

*Voici maintenant des instructions d'opérations logiques :*
*shl : dépile x, décale x à gauche d'un bit dans r, empile r*
*shr : dépile x, décale x à droite d'un bit dans r, empile r*
*and : dépile x, dépile y, r=x and y, empile r*
*or : dépile x, dépile y, r=x or y, empile r*
*xor : dépile x, dépile y, r=x xor y, empile r*
*nor : dépile x, dépile y, r=x nor y, empile r*

Question 5 : que produit la séquence : push M ; push M ; xor ; pop M ?

*Voici par exemple l'instruction « A=B/(C+D) » en langage évolué ; elle peut être traduite par la séquence suivante :*
*push B : on charge d'abord le dividende*
*push C*
*push D*
*add ; le sommet de la pile contient la valeur C+D*
*div ; le sommet de la pile contient B/(C+D)*
*pop A : A ← contenu du sommet de la pile*

L'évolution de la pile pendant l'exécution de cette séquence peut se résumer ainsi :

| | | D | | | |
|---|---|---|---|---|---|
| | C | C | C+D | | |
| B | B | B | B | B/(C+D) | |
| push B | push C | push D | add | div | pop A |

En considérant les différentes opérations effectuées sur la pile, on en déduit une forme particulière (et équivalente) de l'instruction A=B/(C+D) :

| | | | | | |
|---|---|---|---|---|---|
| push B | push C | push D | add | div | pop A |
| B | C | D | + | / | A= |

Cette notation (BCD+/A=) est dite « Polonaise inverse ».

Question 6 : quels peuvent être les avantages et les inconvénients de la notation polonaise par rapport à la notation classique (exemple : A=B/(C+D))?

Question 7 : à quelle instruction en langage évolué correspond la suite d'instructions suivantes :

push B ; push B ; mul ; push =4 ; push A ; mul ; push C ; mul ; sub ; pop D ?

Question 8: quelle est la notation polonaise correspondant à l'instruction de la question précédente (question 7) ?

*Passons maintenant aux instructions de comparaisons et de branchement :*

```
        b ADR       ; branchement inconditionnel à l'adresse donnée
Eti :   nop         ; pseudo-opération de définition d'étiquette
```

*cmp ; dépile deux éléments et les compare arithmétiquement (positionne le registre de condition RC)*
*beq Adr ; branchement à l'adresse indiquée si égalité*
*bne ; branchement à l'adresse indiquée si non égalité*
*blt Adr ; ble Adr ; bgt Adr ; bge Adr ; avec :*
*e pour equal, l pour less, g pour greater et t pour than.*

Question 9 : donner précisément la taille en bits du RC (registre de condition), la signification et le rôle de chaque bit vis-à-vis :
(1) de l'instruction « cmp » de comparaison,
(2) des instructions de branchement « beq », « bne » et « ble ».

*A titre d'exemple, voici une manière de traduire l'instruction « C=max(X,Y) », qui correspond au calcul du maximum de X et Y, le résultat étant placé dans C :*

```
        push X
        push Y
        cmp    ; la comparaison dépile deux éléments
        ble et1 ;si X<=Y aller à et1
        push X ; ici X est > Y : on charge X
        pop C ; et on l'envoie dans C
        b et2 ; c'est fini
et1 :   nop ; ici X<=Y
        push Y ; dans ce cas on charge Y
        pop C ; que l'on envoie dans C
et2 :   nop ; l'étiquette de fin
```

Question 10 : pour le programme ci-dessus, on peut « gagner » une instruction. Montrer comment.

Question 11 : programmer sur MaP le calcul de la valeur absolue de l'entier X. La donnée sera prise en mémoire à l'adresse X et le résultat rangé en B. A la fin du calcul, il ne devra rien rester sur la pile.

Question 12 : X étant un entier strictement positif, programmer sur MaP la séquence d'instruction correspondant à :
Si X est pair
Alors A=X/2 ;
Sinon A=3*X+1 ;

Ici également, à la fin du calcul, il ne devra rien rester sur la pile.

*Voyons maintenant une autre utilisation de la pile. Une pile peut fournir un mécanisme particulièrement puissant pour les transferts de contrôle entre programmes. Dans l'exemple qui suit, la pile de MaP est utilisée pour stocker l'adresse de retour d'un sous-programme (fonction).*
*On définit ainsi l'instruction d'appel de sous-programme :*
*Call Adsp ; Adsp = adresse symbolique de début de sous-programme*
*Dont le microcode peut s'exprimer ainsi :*
*Empile PC (PC est le registre compteur ordinal) PC ← Adsp*

*Un retour du sous-programme est effectué par l'instruction « return » dont le microcode consiste en « dépile PC » (transfert du mot de sommet de pile dans le compteur ordinal (PC).*

Question 13 : écrire le programme de l'exemple et de la question 10 (C=max (X,Y)) sous forme de sous-programme de manière à ce qu'il puisse être appelé à partir d'une instruction « call max », les arguments X et Y étant considérés comme des variables globales et le résultat étant placé dans C.

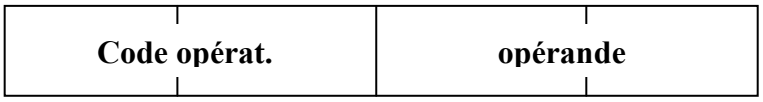
Question 14 : traduire en langage MaP la séquence suivante :
Si ( (C=max(X,Y) > 10 ) aller à Sup
…
Où « max » désigne la fonction de la question précédente.

Question 15 : proposer une technique adaptable à MaP permettant de transmettre des arguments à une fonction (sous-programme), de renvoyer un résultat de fonction au programme appelant.

# Un autre micro processeur fictif (TD2 M2 101)

Principales caractéristiques du µP :

- Mots de 4 octets pour instructions et données
- La mémoire centrale consiste en 100 mots adressables de 00 à 99 (en décimal)
- Un registre ACCU de 4 octets
- Un indicateur « booléen » B de 1 octet qui peut contenir ‘V’ (vrai) ou ‘F’ (faux)
- Un compteur ordinal PC de 2 octets
- Un jeu de 7 instructions de format général :




(2 caractères) (2 chiffres)

Un mot mémoire peut être interprété comme une instruction ou un mot de données.

Le jeu d’instructions est le suivant :

| Code opération | Signification |
|---|---|
| la $x_1x_2$ | ACCU ← [M] |
| ss $x_1x_2$ | M ← [ACCU] |
| ca $x_1x_2$ | Si ACCU= [M]<br>Alors B ← ‘V’<br>Sinon B ← ‘F’ |
| jt $x_1x_2$ | Si B=’V’<br>Alors PC ← M |
| gd $x_1x_2$ | Lire ([m+i]), i=0 ,…,9 |
| pd $x_1x_2$ | Ecrire ([m+i]),<br>i=0 ,…,9 |
| ht $x_1x_2$ | halt |

***Notes** :*

1. *$x_1, x_2 \in \{0,1,….,9\}$*
2. *$M=10* x_1 + x_2$*
3. *$m= 10* x_1$*
4. *Chaque lecture au clavier lit au plus 40 caractères éventuellement complétés par des blancs*
5. *Chaque écriture sur écran affiche une nouvelle ligne de 40 caractères*
6. *La première instruction du programme apparait toujours en 00.*

**Question 1 :**
Faire un schéma représentant le processeur (UC, mémoire, les interactions entre elles et avec les périphériques)

**Question 2 :**
Que produit le petit programme suivant enregistré en mémoire à partir de l'adresse 00 ?
« gd60gd70gd80gd90la80ca90jt27ca91jt27
ca92jt27ca93jt27ca94jt27ca95jt27ca96
jt27ca97jt27ca98jt27ca99jt27pd70ht00
pd60ht00 »

Les lignes entrées au clavier sont :
« mot present »
« mot absent »
« TOTO »
« TITITATATUTUHANSJEANALEXTOTOELLAAUDEIGOR »

*00 : gd60 # lit au plus 40 caractères de la **première** ligne entrée au clavier et les enregistre en mémoire par paquets de 4 de l'adresse 60 à l'adresse 69*
*01 : gd70 # lit au plus 40 caractères de la **deuxième** ligne entrée au clavier et les enregistre en mémoire par paquets de 4 de l'adresse 70 à l'adresse 79*
*02 : gd80 # lit au plus 40 caractères de la **troisième** ligne entrée au clavier et les enregistre en mémoire par paquets de 4 de l'adresse 80 à l'adresse 89*
*….*

**Question 3 :**
Ecrire un programme pour ce processeur qui lit deux mots de 4 caractères au plus ; s'ils sont égaux, afficher le message « les deux mots sont égaux » suivi du mot en question ; sinon afficher le message « les deux mots sont différents » suivi des deux mots.

**Question 4 :**
Afficher les dix premiers entiers à raison d'un par ligne.

**Question 5 :**
Le programme enregistré se limite à l'instruction « gd00 ». Les deux lignes entrées au clavier sont :
« gd00gd80pd80ht00 »
« programme de bootstrap »
Que se passe-t-il lors de l'exécution ?

**Question 6 :**
Ecrire le plus petit programme qui boucle sans fin …

Première série d'exercices
-----===ooOoo===-----

o Enoncé :

Essayer de traduire les deux fonctions qui suivent,
puis comparez avec ce que donne le compilateur de la machine virtuelle PowerPC (lancement par qemu):

```
-----------------------------------------------
int plus(int n) { return n+1; }
int moins(int n) { return n-1; }
-----------------------------------------------
```

o Solution :

1) Sur une station "s" AUTRE que la machine virtuelle,
créer les deux fichiers plus.c et moins.c
à l'aide d'emacs par exemple.
Sur cette même station "s", créer le lanceur, "lanceur.c" :

```
-----------------------------------------------
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
        int n;
        if (argc != 2) perror("usage : lanceur <n>");
        n=atoi(argv[1]);
        printf("%d + 1 : %d\n%d - 1 : %d\n", n, plus(n), n, moins(n));
        exit(EXIT_SUCCESS);
}
-----------------------------------------------
```

A ce stade, lancer les commandes (toujours sur "s") :

```
gcc lanceur.c plus.c moins.c -o lanceur
lanceur 24
...
```

et vérifier le fonctionnement de lanceur.

2) Sur l'émulateur PowerPC :

```
gcc  -S -O plus.c (produit plus.s, source assembleur)
gcc  -S -O moins.c (produit moins.s, source assembleur)
```

Enfin, on appelle gcc pour créer l'exécutable lanceur
(gcc appelle automatiquement ld, l'éditeur des liens):

gcc lanceur.c plus.s moins.s -o lanceur

Et on exécute :

lanceur 12
12 + 1 : 13
12 - 1 : 11

Voici une solution ... :

```
.section __TEXT,__text,regular,pure_instructions
        .section
__TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
.section __TEXT,__text,regular,pure_instructions
        .align 2
        .align 2
        .globl _plus
.section __TEXT,__text,regular,pure_instructions
        .align 2
_plus:
        addi r3,r3,1
        blr
        .subsections_via_symbols


        .section __TEXT,__text,regular,pure_instructions
        .section
__TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
.section __TEXT,__text,regular,pure_instructions
        .align 2
        .align 2
        .globl _plus
.section __TEXT,__text,regular,pure_instructions
        .align 2
_moins:
        addi r3,r3,-1
        blr
        .subsections_via_symbols
```

# TD M 2 101 : Usage gdb

1. Traduire en assembleur le programme distance suivant :

```
int distance(int a, int b)
{
  if(a > b)
    return a-b;
  else
    return b-a;
}
```

2. Ecrire un programme principal de test adist.c:

```
#include <stdio.h>
main ()
{

   printf("appel distance %d\n", distance(100,20));
}
```

3. Tester le programme en assembleur avec le debugger

3.1 En vous aidant du programme en assembleur, écrire dist.c sous la forme :

```
int distance(int a, int b)
{
  asm("subf ….");
  asm("instruction suivante assembleur");….

  asm("mr r3,r0");
}
```

3.2 Lancer l'exécution de ce programme en assembleur avec la commande :

```
$ gcc  -O -g dist.c adist.c -o adist
```

3.3 Lancer le debugger

```
$ gdb adist
>list distance
```

permet d'afficher les numéros des lignes du programme distance en assembleur

```
>break 5
```

met un point d'arrêt ligne 5

```
>run
```

lance l'exécution du programme jusqu'au point d'arrêt

>display a ou x &a (x/d pour avoir la valeur en décimal, x/n pour avoir n adresses consécutives à &a)

affiche le contenu de la variable a ou de l'adresse &a

```
>step ou s
```

exécution pas à pas

```
>info register
```

permet d'afficher le contenu de tous les registres.

> info register ri
permet d’afficher le contenu du registre ri
> define rg
Pour définir un alias visualisant les registres terminé par end
> info register 0
> info register 1
> info register 2
> info register 30
> end
..
>rg
Affiche les contenus des registres définis ci-dessus 0, 1, 2 et 30.

> help
fournit une aide complète
> disas
désassemble le programme, donne la version assembleur de la machine.
> quit
permet de quitter le debugger

NB : on peut utiliser les flèches de rappel.
> list 1,50
...

Deuxième série d'exercices
-----===ooOoo===-----

o Enoncé :

Ecrire et tester les fonctions "maximum", "valeur absolue" et "modulo".

o Solution :

1) Sur une station "s" AUTRE que l'émulateur PowerPC, on définit les 3 fonctions
à l'aide d'emacs.

```
-------------------------------------------------
int max(int a, int b) { return (a>=b)?a:b; }
int abso(int a) { return (a>=0) ? a:-a; }
int modulo(int q, int d) { // q > 0 et d > 0
    return q%d;
}
-------------------------------------------------
```

On regroupe ces fonctions en un fichier "f.c" et,
sur cette même station "s", on crée le lanceur, "lanceur2.c" :

```
-------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
          int a, b;
          if (argc != 3) {
            perror("usage : lanceur2 <a> <b>");
            exit (EXIT_FAILURE);
        }
          a=atoi(argv[1]);
        b=atoi(argv[2]);
          printf("max   ( %d , %d ) : %d\n", a, b, max(a,b));
          printf("modulo( %d , %d ) : %d\n",
           a=abso(a), b=abso(b), modulo(a,b));
          exit(EXIT_SUCCESS);
}
-------------------------------------------------
```

A ce stade, lancer les commandes (toujours sur "s") :

```
gcc lanceur2.c f.c -o lanceur2
lanceur2 24
lanceur2 24 7
lanceur2 24 -7
...
```

et vérifier le fonctionnement de lanceur.

2) Sur la machine virtuelle PPC :

gcc -S -O f.c (produit f.s, source assembleur)

Dans ce fichier "f.s", on repère les codes de max, abso,
et modulo, et on les remplace (progressivement ...) par
"notre" code.

On appelle gcc pour créer l'exécutable lanceur2
(gcc appelle automatiquement ld, l'éditeur des liens):

gcc lanceur2.c f.s -o lanceur2

Et on exécute :

lanceur2 24
lanceur2 24 7
lanceur2 24 -7
...
Et on vérifie.

========== Voici la solution euphor :
==========

```
        .section __TEXT,__text,regular,pure_instructions
        .section __TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
.section __TEXT,__text,regular,pure_instructions
        .align 2
        .align 2
        .globl _max
.section __TEXT,__text,regular,pure_instructions
        .align 2

#
# int max(int a, int b)
# {
#  return (a>=b)?a:b;
# }



_max:
        cmpw cr7,r3,r4
```

```
        bgelr- cr7
        mr r3,r4
        blr
        .align 2
        .globl _abso
.section __TEXT,__text,regular,pure_instructions
        .align 2

#
# int abso(int a)
# {
#  return (a>=0)?a:-a;
# }



_abso:
        srawi r0,r3,31
        xor r3,r0,r3
        subf r3,r0,r3
        blr
        .align 2
        .globl _modulo
.section __TEXT,__text,regular,pure_instructions
        .align 2

#
# int modulo(int q, int d) { // q > 0 et d > 0
#    return q%d;
# }



_modulo:
        divw r0,r3,r4
        mullw r0,r0,r4
        subf r3,r0,r3
        blr
        .subsections_via_symbols
```

# TD M 2 101

Questionnaire à choix multiples – Révision et application des notions de base

1) D'après-vous, pourquoi l'optimiseur du compilateur C essaie-t-il de remplacer les instructions de branchement (saut) par des instructions équivalentes ?

A-pour gagner de la place en mémoire
B-parce qu'un branchement est une instruction qui s'exécute "lentement"
C-pour éviter de "casser" la file d'attente du "pipeline"

2) A quoi sert le registre de liens (Link Register) :

A-à conserver l'adresse de retour dans le programme d'appel, lors d'un appel de sous-programme
B-à éviter un branchement inconditionnel
C-à faciliter l'édition des liens (liaison dynamique)

3) Dans l'instruction "stw r31,8(r1)", l'adressage du deuxième opérande est de type:

A-indirect
B-direct
C-immédiat

4) En exécutant la séquence suivante:

```
li r3,2
xor r4,r4,r4
cmpw cr1,r4,r3
bgt cr1,ETIQUETTE
...
```

A-le programme ne signale aucune erreur et continue l'exécution en séquence (après l'instruction de branchement)
B-le programme ne signale aucune erreur et continue l'exécution à l'adresse ETIQUETTE
C-le programme signale une erreur lors de l'exécution de l'instruction de branchement

5) Le fragment de programme suivant :

```
        cmpwi cr2,r7,0
        bge cr2,suite
        nand r7,r7,r7
        addi r7,r7,1
suite:
        ...
```

A- incrémente le registre 7 de 1 seulement si celui-ci est positif ou nul
B- remplace le registre 7 par son complément à deux incrémenté de 1 seulement si le registre 7 est positif ou nul
C- remplace le contenu du registre 7 par sa valeur absolue.

6) Le fragment de programme suivant :

```
etiq :
        ...
        andi. r0,r9,1
        beq cr0,etiq
        ...
```

A- se poursuit à l'adresse logique "etiq" seulement si le registre 9 contient 0
B- se poursuit en séquence si le registre 9 contient une valeur impaire
C- se poursuit en séquence si le registre 9 contient une valeur paire

7) La séquence suivante :

```
        cmpw cr0,r1,r2
        bgt cr0,etiq
        mr r3,r2
        b suite
etiq:   mr r3,r1
suite:
        ...
```

A- permet de remplacer le contenu du registre 3 par le "maximum" entre R1 et R2
B- permet de remplacer le contenu du registre 3 par le "minimum" entre R1 et R2
C- permet d'échanger les contenus de R1 et R2

8) Le registre 9 contient 10 en base 10. Que contient le registre 3 après exécution de la séquence :

```
        srawi r3,r9,1
        addze r3,r3
```

A- 6 en base 10
B- 7 en base 10

C- 5 en base 10

9) Pour récupérer dans RT le reste de la division entière de RA par RB, on peut utiliser la séquence suivante :

A-
```
divw RT,RB,RA
mullw RT,RT,RA
subf RT,RT,RB
```

B-
```
divw RT,RA,RB
mullw RT,RT,RB
subf RT,RT,RA
```

C-
```
divw RT,RA,RB
mullw RT,RT,RB
subf RT,RA,RT
```

10) La séquence suivante d'instructions :

```
xor r1,r1,r2
xor r2,r1,r2
xor r1,r1,r2
```

A- permet d'échanger les contenus des registres 1 et 2
B- permet de mettre à zéro les registres 1 et 2
C- permet de complémenter les contenus des registres 1 et 2.

---

Syracuse et fonction mystérieuse  exercice sur les boucles

-----===ooOoo===-----

o Enoncé Syracuse :

---

Ecrire et tester en assembleur PowerPC la fonction de Syracuse,
int syr(int n) qui vaut n/2 si n pair (n>0),
 sinon 3n+1
Vérifiez, notamment à l'aide d'un lanceur si cette suite est bornée
ou infinie ...en lançant syr(syr(syr(...syr(n)..)).

o Solution :

---

1-1) en langage C :

 Sur une station "s" AUTRE que la macghuner virtuelle PowerPC (PPC), on définit la fonction
syr(n) à l'aide d'emacs (par exemple) :

---

```
int syr(int n) { return ((n%2) == 0)?n/2:3*n+1; }
```

---

Sur cette même station "s", on crée le lanceur, "lanceur3.c" :

---

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
          int n;
          if (argc != 2) {
            perror("usage : lanceur3 <n>");
            exit (EXIT_FAILURE);
        }
          n=atoi(argv[1]);
          if(n<0)n=-n;

          while(1){
           printf("%d\n",n);
           n=syr(n);
           sleep(1);
          }
          exit(EXIT_SUCCESS);
}
```

---

A ce stade, lancer les commandes (toujours sur "s") :

gcc-3.4 lanceur3.c syr.c -o lanceur3
lanceur3
lanceur3 7
lanceur3 -7
...
et vérifier le fonctionnement du lanceur.

1-2) Sur l'émulateur PPC:

gcc -S -O syr.c (produit syr.s, source assembleur)

Dans ce fichier "syr.s", on repère le code de "syr", on l'étudie ...

```
mr r2,r3
srawi r3,r3,1
addze r3,r3
andi. r0,r2,1
beqlr- cr0
slwi r0,r2,1
add r3,r0,r2
addi r3,r3,1
blr
```

... et on le remplace par
"notre" code, par exemple (à essayer ...) :

```
mr r9,r3
li r4,1
and. r0,r9,r4
beq cr0,suite
slwi r3,r9,1
add r3,r3,r9
addi r3,r3,1
blr
```

On appelle gcc pour créer l'exécutable lanceur3
(gcc appelle automatiquement ld, l'éditeur des liens):

gcc lanceur3.c syr.s -o lanceur3

Et on exécute :

lanceur3
lanceur3 7
lanceur3 -7

...
Et on vérifie.

o Enoncé "fonction mystérieuse" :
---

Le but de cet exercice est de fournir un équivalent C de la fonction assembleur suivante :

```
1           _foncmyst:
2                   li r2,1
3                   cmpw cr7,r2,r3
4                   bgt- cr7,L8
5           L6:
6                   divw r0,r3,r2
7                   mullw r0,r0,r2
8                   subf r0,r0,r3
9                   add r9,r9,r0
10                  addi r2,r2,1
11                  cmpw cr7,r2,r3
12                  ble+ cr7,L6
13            L8:
14                  mr r3,r9
15                  blr
```

1. Isoler et désigner précisément les instructions permettant de calculer un modulo (en C, la fonction " reste " ou " modulo " s'écrit " % "). Justifier.

2. Combien de paramètres comporte cette fonction ? Quel est leur type ? Justifier.

3. Repérer précisément la boucle. Quel registre contrôle son déroulement ? Comment est-il initialisé ? Quelle est la condition d'arrêt ? Comment ce registre évolue-t-il (progression)?

4. La boucle utilise-t-elle des registres de travail ? Si oui, lesquels et pourquoi ?

5. De quel type est le résultat fourni par la fonction et comment est-il calculé ?

6. Donner maintenant un équivalent C de la fonction mystérieuse fournie ci-dessus.

# TD M 2 101 (Programmation assembleur) S2

Analyse et Optimisation de programmes en assembleur

**Exercice 1** Le but est d'optimiser le code source assembleur ci dessous

```
1           _calcul:
2               stw r30,-8(r1)
3               stwu r1,-64(r1)
4               mr r30,r1
5               stw r3,88(r30)
6               stw r4,92(r30)
7               li r0,1
8               stw r0,32(r30)
9               li r0,1
10              stw r0,36(r30)
11          L2:
12              lwz r0,36(r30)
13              lwz r2,92(r30)
14              cmpw cr7,r0,r2
15              ble cr7,L5
16              b L3
17          L5:
18              lwz r2,32(r30)
19              lwz r0,88(r30)
20              mullw r0,r2,r0
21              stw r0,32(r30)
22              lwz r2,36(r30)
23              addi r0,r2,1
24              stw r0,36(r30)
25              b L2
26          L3:
27              lwz r0,32(r30)
28              mr r3,r0
29              lwz r1,0(r1)
30              lwz r30,-8(r1)
31              blr
```

1°) S'agit-il d'un programme ou d'un sous-programme (fonction) ? Pourquoi ?
2°) Combien de paramètres comporte ce (sous-) programme ? Quels sont leurs types et leurs natures (entrée ou sortie) ? Où sont-ils rangés ?
3°) Repérer précisément la boucle en notant les lignes correspondantes. Quelle variable (registre ou case mémoire) contrôle son déroulement ? Comment est-elle initialisée ? Quelle est la condition d'arrêt ? Comment évolue-t-elle ?
4°) La boucle utilise-t-elle des registres de travail ? Si oui, lesquels et pourquoi faire ?
5°) Donner l'équivalent C de ce (sous-) programme. Que calcule-t-il ?
6°) Traduire ce programme C en langage assembleur PowerPC optimisé en supprimant toute instruction de chargement et de déchargement dans la pile. Chaque instruction sera clairement explicitée.
7°) Tester votre programme en suivant les étapes suivantes :

- Ecrire la fonction en C
- Ecrire le programme principal permettant de tester le précédent (gcc *.c maincal.c –o calcul)
- Compiler sur la machine virtuelle PowerPC la fonction en assembleur non optimisé (gcc –S *.c)
- Remplacer les instructions assembleur du fichier *.s par votre programme assembleur et le tester (gcc *.s maincal.c –o calcul)
- Comparer votre version avec la version optimisée du compilateur (gcc –S –o *opt.s –O *.c )

**Exercice 2** Le but est de fournir un équivalent C de la fonction assembleur suivante :

```
1       _apr:
2           li r0,1
3       L2:
4           andi. r2,r4,1
5           beq- cr0,L5
6           mullw r0,r0,r3
7       L5:
8           mullw r3,r3,r3
9           srawi r4,r4,1
10          addze. r4,r4
11          bne+ cr0,L2
12          mr r3,r0
13          blr
```

1°) Détailler pour [R3] = 2 et [R4] = 4, l'exécution pas à pas de cette fonction. Quelle est la valeur retournée en sortie ? Même question pour [R3] = x et [R4] = 9.
2°) En appelant x le paramètre passé dans R3 et n celui passé dans R4, que calcule apr(x,n) ?
3°) Donner l'équivalent C de cette fonction.
4°) Quelles remarques peut on faire sur l'efficacité de cet algorithme ?
5°) Tester votre programme en suivant les mêmes étapes que précédemment :

- Ecrire la fonction en C
- Ecrire le programme principal permettant de tester le précédent (gcc *.c main.c –o puissance)
- Compiler sur machine virtuelle PowerPC la fonction en assembleur optimisé (gcc –S –O *.c)
- Remplacer les instructions assembleur du fichier *.s par votre programme assembleur et le tester (gcc *.s main.c –o puissance)

N.B. 1 : Les divisions entières successives d'un nombre pair par deux donnent toujours au moins un nombre impair parmi les quotients obtenus.
N.B. 2 : srawi RA,RS,SI : Décalage à droite d'un mot RS de SI bits dans le registre RA.

# TD M 2 101 (Programmation assembleur) S2

## Analyse et Optimisation de programmes en assembleur

Un programmeur était chargé d'écrire une fonction mystère. Il est parti en vacances, et son remplaçant a malencontreusement détruit le fichier source C de cette fonction. Tout n'est pas perdu, puisqu'il reste la traduction en assembleur dans mystere.s.

```
1       _mystere:
2               stw r30,-8(r1)
3               stwu r1,-64(r1)
4               mr r30,r1
5               stw r3,88(r30)
6               stw r4,92(r30)
7               stw r5,96(r30)
8               li r0,0
9               stw r0,32(r30)
10              li r0,0
11              stw r0,36(r30)
12      L2:
13              lwz r0,36(r30)
14              lwz r2,96(r30)
15              cmpw cr7,r0,r2
16              blt cr7,L5
17              b L3
18      L5:
19              lwz r0,36(r30)
20              slwi r2,r0,2
21              lwz r0,88(r30)
22              add r2,r2,r0
23              lwz r0,0(r2)
24              slwi r2,r0,2
25              lwz r0,92(r30)
26              add r2,r2,r0
27              lwz r9,32(r30)
28              lwz r0,0(r2)
29              add r0,r9,r0
30              stw r0,32(r30)
31              lwz r2,36(r30)
32              addi r0,r2,1
33              stw r0,36(r30)
34              b L2
35      L3:
36              lwz r0,32(r30)
37              mr r3,r0
38              lwz r1,0(r1)
39              lwz r30,-8(r1)
40              blr
```

Il reste également le programme de test, principal.c :

```
1       extern int mystere(int ind[], int tab[], int n);
2       int main()
3       {
4               int t[7] = { 10,20,30,40,50,60,70 };
5               int i[3] = { 0,3,4 };
6               int resultat, j;

7                       for (j=0; j<1000000 ; j++)
8                               resultat = mystere(i,t,3);
9                       printf( "resultat = %d \n", resultat);
10      }
```

**Questions :**

1. A quoi voit-on qu'il s'agit d'une fonction feuille ?
2. Reconstituez l'algorithme de la fonction. Ecrivez la fonction en C.
3. Traduire cette fonction en assembleur.
4. Compiler la fonction en assembleur non optimisé, dans le but de comparer les performances avec votre version personnelle. Pour distinguer les différents temps d'exécution, on utilise la commande time et on appelle N fois (N=1000000 dans le programme principal précédent) la même fonction dans le programme principal.
5. Remplacez la partie non optimisée par votre version. Comparer les performances.
6. Compiler la fonction en assembleur optimisé. Comparer la solution obtenue avec votre version, puis comparer les performances (êtes-vous meilleur optimiseur que le compilateur gcc ?).

# Jeu d'instructions Assembleur PowerPc

## Notations

| | |
|---|---|
| $D$ | le déplacement |
| $CR$ | le Registre de Condition |
| Rx | registre entre r0 et r31 |
| | (RT pour "Target", RS pour "Source", RA et RB pour des opérandes) |
| $(Rx)$ | le contenu du registre Rx |
| $(RA\|0)$ | si RA=0, alors 0, sinon le contenu de RA |
| $V, SI$ | quantité immédiate signée sur 16 bits |
| $UI$ | quantité immédiate non signée sur 16 bits |
| $rep(N, B)$ | champ formé de $N$ répétitions du bit $B$ |
| $mem(A, N)$ | $N$ octets en mémoire à partir de l'adresse $A$ |
| $A\|\|B$ | Le champ obtenu par concaténation de $A$ et $B$ |
| $exts(V)$ | la valeur $V$ avec extension de signe sur 32 bits |
| $A\{i \dots j\}$ | les bits d'indice $i$ à $j$ de $A$ |
| $A \leftarrow B$ | affectation |



## Mouvements de données

### Par octet

**Chargement octet et zéro**

`lbz RT,D(RA)` $RT \leftarrow rep(24, 0)||mem((RA|0) + D, 1)$

**Chargement octet et zéro avec mise-à-jour**

`lbzu RT,D(RA)` $RT \leftarrow rep(24, 0)||mem((RA) + D, 1)$;
$RA \leftarrow (RA) + D$

**Chargement indexé octet et zéro**

`lbzx RT,RA,RB` $RT \leftarrow rep(24, 0)||mem((RA|0) + (RB), 1)$

**Chargement indexé octet et zéro avec mise-à-jour**

`lbzux RT,RA,RB` $RT \leftarrow rep(24, 0)||mem((RA) + (RB), 1)$
$RA \leftarrow (RA) + (RB)$

**Rangement octet**

`stb RS,D(RA)` $mem((RA|0) + D, 1) \leftarrow (RS)24..31$

**Rangement octet avec mise-à-jour**

`stbu RS,D(RA)` $mem((RA) + D, 1) \leftarrow (RS)24..31$
$RA \leftarrow (RA) + D$

**Rangement indexé octet**

`stbx RS,RA,RB` $mem((RA|0) + (RB), 1) \leftarrow (RS)24..31$

**Rangement indexé octet avec mise-à-jour**

`stbux RS,RA,RB` $mem((RA) + (RB), 1) \leftarrow (RS)24..31$
$RA \leftarrow (RA) + (RB)$

**Par demi-mot**

**Chargement demi-mot et zéro**

`lhz RT,D(RA)` $RT \leftarrow rep(16,0)||mem((RA|0)+D,2)$

**Chargement demi-mot et zéro avec mise-à-jour**

`lhzu RT,D(RA)` $RT \leftarrow rep(16,0)||mem((RA)+D,2)$
$RA \leftarrow (RA)+D$

**Chargement indexé demi-mot et zéro**

`lhzx RT,RA,RB` $RT \leftarrow rep(16,0)||mem((RA|0)+(RB),2)$

**Chargement indexé demi-mot et zéro avec mise-à-jour**

`lhzux RT,RA,RB` $RT \leftarrow rep(16,0)||mem((RA)+(RB),2)$
$RA \leftarrow (RA)+(RB)$

**Chargement algébrique demi-mot**

`lha RT,D(RA)` $RT \leftarrow exts(mem((RA|0)+D,2))$

**Chargement algébrique demi-mot avec mise-à-jour**

`lhau RT,D(RA)` $RT \leftarrow exts(mem((RA)+D,2))$
$RA \leftarrow (RA)+D$

**Chargement algébrique indexé demi-mot**

`lhax RT,RA,RB` $RT \leftarrow exts(mem((RA|0)+(RB),2))$

**Chargement algébrique indexé demi-mot avec mise-à-jour**

`lhaux RT,RA,RB` $RT \leftarrow exts(mem((RA)+(RB),2))$
$RA \leftarrow (RA)+(RB)$

**Rangement demi-mot**

`sth RS,D(RA)` $mem((RA|0)+D),2) \leftarrow (RS)16..31$

**Rangement demi-mot avec mise-à-jour**

`sthu RS,D(RA)` $mem((RA)+D),2) \leftarrow (RS)16..31$
$RA \leftarrow (RA)+D$

**Rangement indexé demi-mot**

`sthx RS,RA,RB` $mem((RA|0)+(RB),2) \leftarrow (RS)16..31$

**Rangement indexé demi-mot avec mise-à-jour**

`sthux RS,RA,RB` $mem((RA)+(RB),2) \leftarrow (RS)16..31$
$RA \leftarrow (RA)+(RB)$



**Par mot**

**Chargement mot**

`lwz RT,D(RA)` $RT \leftarrow mem((RA|0)+D,4)$

**Chargement mot avec mise-à-jour**

`lwzu RT,D(RA)` $RT \leftarrow mem((RA)+D,4)$
$RA \leftarrow (RA)+D$

**Chargement indexé mot**

`lwzx RT,RA,RB` $RT \leftarrow mem((RA|0)+(RB),4)$

**Chargement indexé mot avec mise-à-jour**

`lwzux RT,RA,RB` $RT \leftarrow mem((RA)+(RB),4)$
$RA \leftarrow (RA)+(RB)$

**Rangement mot**

`stw RS,D(RA)` $mem((RA|0)+D,4) \leftarrow (RS)$

**Rangement mot avec mise-à-jour**

`stwu RS,D(RA)` $mem((RA)+D,4) \leftarrow (RS)$
$RA \leftarrow (RA)+D$

**Rangement indexé mot**

`stwx RS,RA,RB` $mem((RA|0)+(RB),4) \leftarrow (RS)$

**Rangement indexé mot avec mise-à-jour**

`stwux RS,RA,RB` $mem((RA)+(RB),4) \leftarrow (RS)$
$RA \leftarrow (RA)+(RB)$

**Chargement de constantes**

**Chargement immédiat**

`li RT,V` $RT \leftarrow exts(V)$

**Chargement immédiat en partie haute**

`lis RT,V` $RT(0..15) \leftarrow V$

En fait ces deux instructions sont des mnémoniques étendus qui représentent `addi RT,0,V` et `addis RT,0,V`

**Mouvement de registre à registre**

**Chargement registre général**

`mr RT,RS` $RT \leftarrow (RS)$
C'est un mnémonique étendu pour `or RT,RS,RS`

**Chargement à partir du registre de lien (move from link register)**

`mflr RT` $RT \leftarrow (LR)$

**Rangement dans registre de lien (move to link register)**

`mtlr RS` $LR \leftarrow (RS)$

## Branchements et conditions

**Branchement inconditionnel (goto ou jump)**

`b adr`

**Branchement inconditionnel avec lien (adresse suivante dans LR, Link Register)**

`bl adr`

**Branchement conditionnel**

| | |
|---|---|
| `bxx CR,adr` | Avec *xx* défini ci-dessous |

**Branchement conditionnel avec lien**

| | |
|---|---|
| `bxxl CR,adr` | Avec *xx* défini ci-dessous |

**Branchement conditionnel au registre de lien (LR)**

| | |
|---|---|
| `bxxlr CR` | Avec *xx* défini ci-dessous |

**Branchement conditionnel au registre de lien avec lien**

| | |
|---|---|
| `bxxlrl CR` | Avec *xx* défini ci-dessous |

**Code *xx* de branchement et signification**

| | |
|---|---|
| `lt` | *less than* $:<$ |
| `eq` | *equal to* $:=$ |
| `gt` | *greater than* $:>$ |
| `le` | *less than or equal to* (*not greater than*) $:<=$ |
| `ge` | *greater than or equal to* (*not less than*) $:>=$ |
| `ne` | *not equal to* $:!=$ |
| `nl` | *not less than* $:>=$ |
| `ng` | *not greater* $:<=$ |
| `z` | *zero* |
| `nz` | *not zero* |

Les prédictions de branchements : les deux suffixes + et − peuvent être ajoutés à un code mnémonique de branchement conditionnel :

1. + indique que la prédiction de branchement est celle prévue
2. − indique que la prédiction de branchement n'est pas celle prévue

## Opérations arithmétiques et logiques

La notation `[.]` signifie que le mnémonique peut être suivi d'un point. Dans ce cas, l'opération met à jour le sous-registre de condition CR0.

**Opérations arithmétiques**

| | |
|---|---|
| `addi RT,RA,SI` | $RT \leftarrow (RA\|0) + exts(SI)$ |
| `addis RT,RA,SI` | $RT \leftarrow (RA\|0) + (SI)\|\|rep(16,0)$ |
| `add[.] RT,RA,RB` | $RT \leftarrow (RA) + (RB)$ |
| `subf[.] RT,RA,RB` | $RT \leftarrow (RB) - (RA)$ |
| `neg[.] RT,RA` | $RT \leftarrow -(RA)$ |
| `mulli RT,RA,SI` | $RT \leftarrow (RA) * exts(SI)$ |
| `mullw[.] RT,RA,RB` | $RT \leftarrow (RA) * (RB)$ |
| `divw[.] RT,RA,RB` | $RT \leftarrow (RA)/(RB)$ |

**Note :** la séquence suivante permet de calculer le reste d'une division entière :



```
divw  RT,RA,RB
mullw RT,RT,RB
subf  RT,RT,RA
```

Les comparaisons positionnent les indicateurs d'un sous-registre de condition.

**Comparaisons**

| | |
|---|---|
| `cmpwi CR,RA,SI` | comparaison de $(RA)$ et $exts(SI)$, résultat ds CR |
| `cmpw CR,RA,RB` | comparaison de $(RA)$ et $(RB)$ |

**Opérations logiques**

| | |
|---|---|
| `andi RA,RS,UI` | $RA \leftarrow (RS)\ and\ rep(16,0)\|\|UI$ |
| `ori RA,RS,UI` | $RA \leftarrow (RS)\ or\ rep(16,0)\|\|UI$ |
| `xori RA,RS,UI` | $RA \leftarrow (RS)\ xor\ rep(16,0)\|\|UI$ |
| `andis RA,RS,UI` | $RA \leftarrow (RS)\ and\ UI\|\|rep(16,0)$ |
| `oris RA,RS,UI` | $RA \leftarrow (RS)\ or\ UI\|\|rep(16,0)$ |
| `xoris RA,RS,UI` | $RA \leftarrow (RS)\ xor\ UI\|\|rep(16,0)$ |
| `and[.] RA,RS,RB` | $RA \leftarrow (RS)\ and\ (RB)$ |
| `or[.] RA,RS,RB` | $RA \leftarrow (RS\ )or\ (RB)$ |
| `xor[.] RA,RS,RB` | $RA \leftarrow (RS)\ xor\ (RB)$ |
| `nand[.] RA,RS,RB` | $RA \leftarrow (RS)\ nand\ (RB)$ |
| `nor[.] RA,RS,RB` | $RA \leftarrow (RS)\ nor\ (RB)$ |
| `eqv[.] RA,RS,RB` | $RA \leftarrow (RS)\ eqv\ (RB)(eqv = notxor)$ |
| `andc[.] RA,RS,RB` | $RA \leftarrow (RS)\ and\ not(RB)$ |
| `orc[.] RA,RS,RB` | $RA \leftarrow (RS)\ or\ not(RB)$ |

**Extension de signe**

| | |
|---|---|
| `extsb[.] RA,RS` | $RA \leftarrow exts(RS24..31)$ |
| `extsh[.] RA,RS` | $RA \leftarrow exts(RS16..31)$ |

**Décalages**

| | |
|---|---|
| `slw[.] RA,RS,RB` | $RA \leftarrow (RS) << (RB)$ |
| `srw[.] RA,RS,RB` | $RA \leftarrow (RS) >> (RB)$ |
| `sraw[.] RA,RS,RB` | $RA \leftarrow exts(RS0..31 - (RB))$ (décalage algébrique) |
| `srawi[.] RA,RS,SI` | $RA \leftarrow exts(RS0..31 - SI)$ |